Rapid Generation of High-Quality RISC-V Processors from Functional Instruction Set Specifications

Gai Liu[1,2], Joseph Primmer[1], Zhiru Zhang[1]
[1]School of Electrical and Computer Engineering, Cornell University, Ithaca, NY [2]Xilinx, Inc., San Jose, CA
gai.liu@xilinx.com, {jp2228, zhiruz}@cornell.edu

ABSTRACT

The increasing popularity of compute acceleration for emerging domains such as artificial intelligence and computer vision has led to the growing need for domain-specific accelerators, often implemented as specialized processors that execute a set of domain-optimized instructions. The ability to rapidly explore (1) various possibilities of the customized instruction set, and (2) its corresponding micro-architectural features is critical to achieve the best quality-of-results (QoRs). However, this ability is frequently hindered by the manual design process at the register transfer level (RTL). Such an RTL-based methodology is often expensive and slow to react when the design specifications change at the instruction-set level and/or micro-architectural level.

We address this deficiency in domain-specific processor design with ASSIST, a behavior-level synthesis framework for RISC-V processors. From an untimed functional instruction set description, ASSIST generates a spectrum of RISC-V processors implementing varying micro-architectural design choices, which enables effective tradeoffs between different QoR metrics. We demonstrate the automatic synthesis of more than 60 in-order processor implementations with varying pipeline structures from the RISC-V 32I instruction set, some of which dominate the manually optimized counterparts in the area-performance Pareto frontier. In addition, we propose an autotuning-based approach for optimizing the implementations under a given performance constraint and the technology target. We further present case studies of synthesizing various custom instruction extensions and customized instruction sets for cryptography and machine learning applications.

1 INTRODUCTION

The end of Dennard scaling has led to a rapid growth of hardware accelerators to meet the ever more stringent performance and energy requirements [4, 8]. While traditional hardware specialization primarily focused on fixed-function circuits, programmable accelerators in the form of domain-specific microprocessors are becoming increasingly popular (e.g., the Pixel visual core [21], Google TPU [13] and Microsoft's Catapult project [18]). Such programmable accelerators deliver superior performance than the general-purpose processors while still maintaining software programmability. Additionally, the advent of open-sourced ISAs further catalyzes the rise of programmable accelerators. Such open-sourced ISAs, most notably the RISC-V ISA [3], enable standardized and extensible frameworks for fast deployment of low-cost, customizable processors.

When design these programmable accelerators, the traditional register-transfer-level (RTL) methodology is time consuming and error prone. Designers must wrestle with low-level hardware description languages to explore the large design space of the accelerators, which is particularly challenging due to the formidably high cost of implementing and verifying multiple ISA-level and micro-architectural level design choices at the register transfer level. This work focuses on productive hardware specialization for programmable architectures. We propose ASSIST[1], a synthesis framework for generating high-quality and customizable RISC-V processors from functional instruction set specifications. Contrary to configurable cores (e.g., Xtensa [11] and FabScalar [6]), where the structure of the processor pipeline is restricted to a few pre-defined configurations, ASSIST generates a large set of design points with varying pipeline structures and optimized hazard resolution mechanisms.

The focus of this work is not on the automatic synthesis of sophisticated micro-architectural level mechanisms; instead, we target the fast generation of domain-specific RISC-V cores with user-defined instruction extensions. We limit the scope to synthesizing single-issue, in-order, pipelined processors; potential micro-architectural level extensions are discussed in Section 6. Our major technical contributions are:

- We design an embedded domain-specific language in Python for specifying the functional instruction set.
- We propose synthesis techniques to automatically infer resource-efficient datapath implementations from the user-defined instruction set, and generate the optimal forwarding and hazard resolution logic.
- We present an autotuning-based design space explorer which automatically optimizes the pipeline microarchitecture based on QoR metrics of the design space.
- We demonstrate the automatic synthesis of more than 60 processor implementations from the single source of the RISC-V 32I instruction set architecture, some of which outperform widely-used manual designs.
- We study the synthesis of RISC-V processors with custom instruction extensions for cryptography and machine learning applications, which achieves up to 9X performance improvements over the baseline general-purpose RISC-V processors.

The rest of the paper is organized as follows: Section 2 motivates our approach; Section 3 describes the synthesis techniques in ASSIST; Section 4 details experimental results; Section 5 discusses the related work, followed by conclusions in Section 6.

2 MOTIVATIONS FROM EXISTING C-BASED HIGH-LEVEL SYNTHESIS TOOLS

C-based high-level synthesis (HLS) is a popular approach to generate RTL hardware designs from software descriptions [9]. Such HLS tools generally first parse the input software program into a control-data flow graph (CDFG), then apply hardware-specific transformations to synthesize the hardware datapath and the corresponding control logic, typically in the form of finite state machines (FSMs). While this approach is efficient for fixed-function circuits, it often leads to sub-optimal designs when targeting programmable architectures.

As a case study, we synthesize the MIPS processor design from the CHStone HLS benchmark suite [12] using a state-of-the-art commercial HLS tool [20]. Figure 1 shows the code snippet of the C code used as the input to the HLS tool. We make the following observations from the quality-of-results (QoR) of the HLS-synthesized design:

- **Sub-optimal initiation interval**. Existing HLS tools usually rely on compile-time dependency analysis to infer the lowest

DAC '19, June 2–6, 2019, Las Vegas, NV, USA

ACM ISBN 978-1-4503-6725-7/19/06…$15.00
https://doi.org/10.1145/3316781.3317890

[1]ASSIST stands for *architectural synthesis system for instruction set targets*.

```
while (pc != 0) {
  #pragma HLS pipeline
  ins = imem[IADDR (pc)]; pc = pc + 4; op = ins >> 26;
  switch (op) {
    case R:
      funct = ins & 0x3f; shamt = (ins >> 6) & 0x1f;
      rd = (ins >> 11) & 0x1f; rt = (ins >> 16) & 0x1f;
      rs = (ins >> 21) & 0x1f;
      switch (funct) {
        case ADDU: reg[rd] = reg[rs] + reg[rt]; break;
        /* additional funct omitted */
      } break;
    case JAL:
      tgtadr = ins & 0x3ffffff; reg[31] = pc;
      pc = tgtadr << 2; break;
    /* additional cases omitted */
    default:
      address = ins & 0xffff;
      rt = (ins >> 16) & 0x1f; rs = (ins >> 21) & 0x1f;
      switch (op) {
        case LUI: reg[rt] = address << 16; break;
        /* additional op omitted */
        default: pc = 0; break;
      } break;
}}
```

Figure 1: Code snippet of the MIPS processor benchmark [12] for high-level synthesis.

achievable initiation interval that guarantees functional correctness. However, compile-time analysis is inherently conservative, which can potentially prevent the synthesized designs from achieving the best pipeline throughput. In this example, the HLS synthesized design achieved an initiation interval of four due to the loop-carried dependency on the program counter pc.

- **Inefficient datapath implementation**. Different instructions specified using case statements are compiled into different basic blocks of the CDFG. Current HLS tools have insufficient support for exploring the intricate tradeoff between the number of functional units and the complexity of multiplexer networks. Consequently, HLS tools tend to allocate duplicated resources even though the operations can share the same resource. In this study, the HLS tool allocates two 32-bit physical multipliers for the signed and unsigned multiplication operations, respectively, while only one multiplier is needed in a resource-efficient implementation.
- **Complex FSM control logic**. The HLS tool generates a 7-state FSM to control the pipeline execution, leading to additional resource usage. Compared to an optimized processor implementation without explicit control FSM, the FSM logic in the HLS generated design incurs additional resource usage and increases cycle time.

3 ASSIST TECHNIQUES

ASSIST combines domain-specific language design with intelligent synthesis algorithms to enable efficient processor implementation. There are three major components in the ASSIST synthesis framework: (1) The instruction specification interface (Section 3.1) employs an architecture description language (ADL) embedded in Python to allow designers to explicitly specify the functional behavior of the instructions. (2) The architectural synthesis engine (Section 3.2) generates an optimized datapath and control logic based on the information from the ADL. (3) The pipelining schedule autotuner (Section 3.3) autotunes the pipeline schedule to optimize the QoR of the final design.

Figure 2 provides a high-level view of the ASSIST synthesis flow. The architectural synthesis engine takes the instruction set specification as well as a pipeline schedule describing the number of stages and locations of the pipeline registers, and generates the microprocessor implementation in the form of a Chisel RTL. The design QoR is then measured by RTL simulation (using representative benchmarks) and physical implementation. These measured QoR metrics are used in turn as the input to the pipeline schedule autotuner to iteratively optimize the pipeline schedule.

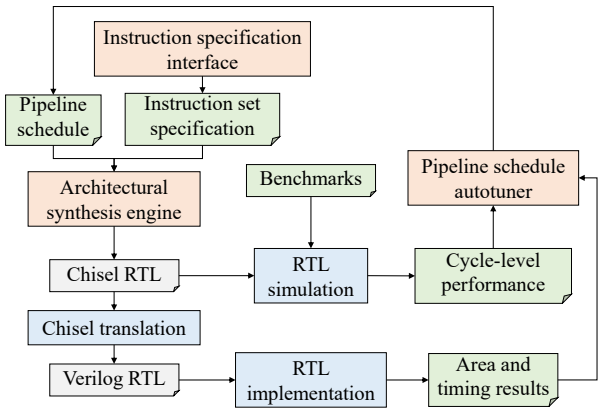

Figure 2: The overall flow of ASSIST.

3.1 Instruction Specification Interface

The embedded ADL used in the instruction specification interface is the key to enable efficient datapath implementation. To designers, the semantics of the ADL is rich enough to support a large range of operations; it also allows the designers to concisely describe changes to the instruction set (e.g., removing expensive instructions from the base instruction set, or changing the semantics of custom instructions) at the behavior level without worrying about the low-level implementation details. In the meantime, the language constructs are specially designed such that many important micro-architectural level design decisions can be inferred from the ADL. To achieve these goals, our ADL is constructed around the notion of micro-operations (micro-ops), which are atomic operations used to formulate the more complex behavior in a complete instruction.

```
def execute_add():
  create_inst('ADD'); tmp = add(rs1, rs2)
  assign(tmp, rd); inc_pc()

def execute_beq():
  create_inst('BEQ'); tmp = cmp_ne(rs1, rs2)
  update_pc_with_pred(tmp, pc, imm_b)

def execute_lb():
  create_inst('LB'); addr = add(rs1, imm_i)
  mem_read(addr, 1, rd, SIGNED); inc_pc()

def execute_simd_add():
  create_inst('SIMD_ADD')
  def kernel():
    rs1_l = bit_range(rs1, 15, 0); rs1_h = bit_range(rs1, 31, 16)
    rs2_l = bit_range(rs2, 15, 0); rs2_h = bit_range(rs2, 31, 16)
    sum_l = add(rs1_l, rs2_l); sum_h = add(rs1_h, rs2_h)
    sum_val = bit_concat(sum_h, sum_l)
    return sum_val
  sum_val = compute_kernel(kernel, rs1, rs2)
  assign(sum_val, rd); inc_pc()
```

Figure 3: Examples of instruction specifications in ASSIST.

Figure 3 gives three examples of specifying the RISC-V 32I instructions in the ADL: `add`, `beq`, and `lb`. In addition, the `simd_add` instruction, which implements two independent 16-bit additions, is used to demonstrate how customized instructions are defined. Following the RISC-V ISA specification, we allow each instruction to take at most two input operands, and to produce at most one output operand. In Figure 3, `rs1`, `rs2`, `rd`, `imm_i`, and `imm_b` are user-defined constructs representing the register specifiers and immediate fields, respectively. Micro-ops such as `create_inst`, `add`, `bit_range`, and `mem_read` are used to compose a sequence of actions or function calls (e.g., the `kernel` function in instruction `simd_add`) that operate on the input operands, and generate the output operand and/or update the architectural state. The output bitwidths of the arithmetic and logic type micro-ops are automatically inferred from their input operands. Table 1 details the available micro-ops in the current version of ASSIST.

The benefits of using micro-ops are twofold. Firstly, it helps create a clear datapath-control split where the semantics of the

Table 1: Micro-ops in the instruction specification interface.

Function	Description
create_inst	Create instruction with name and encoding
inc_pc	Increment PC to the next word
update_pc	Update the value of PC with address
update_pc_with_pred	Update PC if pred is true; otherwise inc_pc
add, sub, xor, or, and	Arithmetic operations
slt/sltu	Set true if less than (signed/unsigned)
sll	Logical left shift
srl, sra	Logical/arithmetic right shift
bit_range	Select bit slice of certain range
bit_concat	Concatenate two bit slices
select	Select true or false branch based on condition
cmp_(eq/ne)	True if operands are equal/not equal
cmp_(lt/ge/gt)_(s/u)	Signed/unsigned comparisons
mem_read/write	Read/write certain bytes from/to memory
assign	Assign operand to another value

micro-ops indicate whether one micro-op should be implemented in the datapath or the control logic. For example, compute type micro-ops such as `add` and `bit_range` are intuitively mapped to datapath, while control type micro-ops such as `update_pc` are assigned to the control logic. Secondly, the use of pre-defined micro-ops streamlines the resource sharing among different instructions. That is, since different instructions are executed at different time stamps in a microprocessor pipeline, the micro-ops used in multiple instructions can be shared in the datapath, while the dedicated control logic (Section 3.2.2) resolves structural hazards.

3.2 Architectural Synthesis Engine

The architectural synthesis engine translates the instruction set specification into the processor implementation in RTL, which involves two major steps. (1) Datapath synthesis populates the pre-defined combinational functional units using instruction-set-specific information. Pipeline registers are then inserted between the functional units based on the pipeline schedule (either from the autotuner detailed in Section 3.3, or manually provided). (2) The control logic is inferred, which resolves data and control hazards through data forwarding, pipeline stalling, and control speculation.

3.2.1 Datapath Synthesis

ASSIST abstracts the datapath of RISC-V processors into the frontend and backend, both of which are further decomposed into finer-grain functional units. These functional units are logically separated building blocks of the processor datapath, which are pre-defined and parameterized templates to be instantiated with the information from the input instruction set.

Datapath frontend. The *(a) instruction fetch unit* calculates the next PC value based on the different modes of branch and jump instructions inferred from the instruction set specification, specifically from the *update_pc* micro-op. The *(b) instruction decoder* generates decoded register addresses and immediate fields from the instruction encoding based on the user's annotation. The *(c) register read* module instantiates the register file, and reads out the register values based on the register source addresses from the decoder. In addition, the forwarding path leading to the register file is handled in this module by introducing a multiplexer after the register output, which selects either the data from the register or the forwarding path selected by the forwarding control logic.

Datapath backend. The *(d) branching unit* computes the predication whether a branch is taken, and calculates the corresponding branch target based on the `compare` type micro-ops and the addressing operations in the instruction set. The *(e) ALU operand select logic* chooses the correct operand sources for each instruction. In addition, it handles the forwarding path from ALU output or writeback module to the inputs of the ALU with a multiplexer at the output of the operand select module. The *(f) ALU logic* executes the various arithmetic operations in the instruction set defined using the compute type micro-ops. The ALU datapath is implemented by fusing the common micro-ops across different instructions. This is achieved by finding the minimum cover of all the used compute type micro-ops across the instructions. The control logic selects the correct multiplexer signals based on the instruction decoder. Such an approach naturally maximizes resource sharing within the ALU, since the minimum cover guarantees that no duplicated implementations of the micro-ops exist in the ALU datapath. The *(g) memory request and register writeback* module handles memory requests and responses using a latency-insensitive interface. The writeback multiplexer selects the source for writing back to the register file based on the register assignment micro-op.

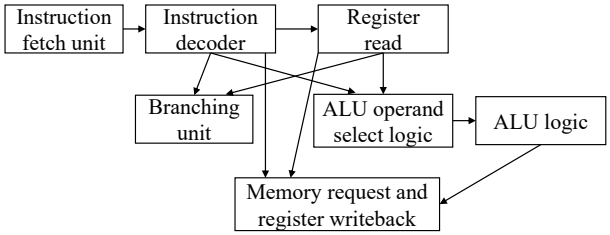

Figure 4: Datapath functional units and their data dependencies.

Pipelining. These seven datapath functional units (from *a* to *g*) constitute a data flow graph with edges representing data dependency between the functional units. Figure 4 sketches such data dependencies between the functional units. Pipelining is the process of adding schedule information to this data flow graph by grouping the functional units into different pipeline stages. A *pipeline schedule* is a vector specifying the position of each functional unit in the pipeline. For instance, a pipeline schedule of `[1111111]` represents a combinational datapath with all seven functional units scheduled in stage one. The pipeline schedule, `[1234567]`, is a highly pipelined design with each functional unit occupying a separate pipeline stage. Given an input pipeline schedule, the architectural synthesis engine analyzes the use-define relationship among the functional units, and automatically inserts pipeline registers to relay data across pipeline boundaries. ASSIST also supports pipelining within a functional unit, which is typically needed within the *ALU logic* if long delay paths exist due to complex custom instructions. We support pipelining a functional unit by adding pipeline registers to the front of the functional unit, and enabling retiming in the downstream logic synthesis tool. With cycle time targeting retiming, the locations of the registers within a combinational module are automatically optimized to achieve the optimal clock frequency [14].

3.2.2 Control Logic Generation

It is often challenging, even in manual designs, to devise the optimal set of pipeline control signals that maximize the pipeline throughput while ensuring the correct pipeline execution. ASSIST solves this problem by proposing a set of rule-based techniques to automatically generate the pipeline forwarding, speculation, stalling, and squash signals. For data hazards, the pipeline control logic enables data forwarding whenever possible, and stalls the pipeline if necessary. For control hazards, the pipeline control logic issues pipeline squash signal when a branch is mis-predicted, and resumes the execution from the correct branch target. Specifically, the *forwarding control logic* is responsible for determining the select signal of the forwarding multiplexers. For each of the possible forwarding paths in the datapath, ASSIST generates a control signal that activates the forwarding path if the register writeback address matches that of the data consuming instruction(s). The *data hazard stall logic* stalls the pipeline by inserting bubbles into the pipeline whenever a data hazard cannot be resolved by forwarding. For branches that are mistakenly predicted, the `pipeline squash logic` is asserted to squash all instructions in the mis-predicted path.

3.3 Pipeline Schedule Autotuner

It is generally difficult to obtain accurate QoR estimates at the pre-RTL synthesis stage for several reasons: (1) the synthesized processor pipeline contains complex control logic that complicates

static timing analysis, which usually cannot be accurately predicted without detailed implementation. (2) multiple supported technology targets make timing pre-characterization expensive and inflexible for new technology targets. (3) many important QoR metrics such as execution time and area are often workload dependent, and are significantly impacted by scheduling. Such metrics cannot be accurately determined statically. The pipeline schedule autotuner iteratively invokes the downstream CAD tool to obtain more accurate QoR estimates and uses the information to guide an intelligent autotuning-based design space exploration. Specifically, ASSIST integrates OpenTuner [1] as the autotuning engine. OpenTuner is an iterative meta-heuristic based autotuning framework, which maintains an ensemble of search techniques. At each iteration, OpenTuner uses the multi-armed bandit algorithm to select one of the search techniques for proposing the new design point to explore.

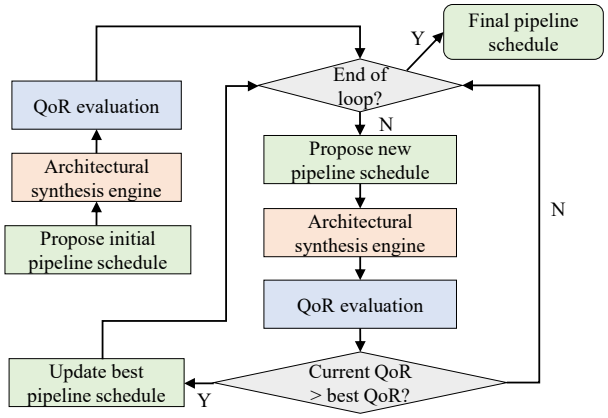

Figure 5: Flow chart of the autotuning process.

Figure 5 illustrates the overall autotuning process in ASSIST. Starting with an initial pipeline schedule, the architectural synthesis engine generates the RTL implementation, and uses the CAD implementation tool to measure its QoR. The design point, together with the measured QoR, is fed into the autotuner. The autotuner then proposes and evaluates a new pipeline schedule. This process iterates until reaching the desired QoR or a pre-determined runtime budget. In addition to optimizing for a single objective, the pipeline schedule autotuner also supports a constrained optimization target, where design points that fail the user-specified constraints are immediately discarded during the search process. The best pipeline schedule found during the autotuning process is returned as the final pipeline schedule.

4 EXPERIMENTAL RESULTS

We first present the result of the design space of ASSIST-generated designs targeting the RISC-V 32I ISA. We then study the cycle time constrained performance optimization using the pipeline schedule autotuner, followed by discussions on integrating instruction and data memories to the processor pipeline. Finally, we present case studies of supporting custom instruction extensions and domain-specific instruction set in ASSIST.

4.1 Design Space Exploration

ASSIST can be used to synthesize and evaluate a large number of design points for a given instruction set target by enumerating the feasible pipeline schedules. We benchmark the QoR of the design points found by ASSIST against three manually optimized designs in the Sodor Processor Collection [7]. The Sodor Processor Collection provides the optimized implementations of 1-, 3-, and 5-stage single-issue, in-order processors of the RISC-V 32I ISA. In this experiment, we use ASSIST to enumerate 64 design points with different pipeline schedules ranging from a combinational datapath to a 7-stage pipelined implementation. For the Virtex-7 FPGA target, we implement both the manual designs and the ASSIST-generated designs using Vivado version 2017.1. For the ASIC target, we implement the designs using Synopsys Design Compiler and IC Compiler II version 2016.12 targeting a 32nm technology library.

Table 2: Top three designs with different optimization targets using a 32nm ASIC technology library — `Top-3`: the top-three designs with the highest QoR of the corresponding target; `Base`: manual 1-, 2-, 5-stage design; `CT`: clock period in nanosecond after implementation; `Area`: design area in μm^2; `RT`: total runtime in millisecond over seven kernels.

Optimization target	Schedule	CT	Area	RT
	Base-1	0.91	47908	0.625
	Base-2	0.34	52202	0.267
	Base-5	0.32	51706	0.292
Cycle time	[1112233]	*0.33*	54530	0.330
	[1223344]	*0.33*	56320	0.362
	[1223334]	*0.33*	53956	0.301
Area	[1111111]	0.90	*48964*	0.618
	[1222222]	0.36	*51692*	0.283
	[1122222]	0.38	*51874*	0.298
Execution time	[1111222]	0.36	53283	*0.247*
	[1111122]	0.38	53587	*0.261*
	[1112222]	0.34	52281	*0.267*

Figure 6 plots the three-dimensional design space of the QoR of the implemented designs in terms of area, cycle time and execution time. The total execution time is measured as the runtime for completing a set of seven representative benchmarks from the RISC-V benchmark suite, which include sorting algorithms, sparse matrix computation, recursive algorithms, and so on. The red square dots in Figure 6 represent the three manual designs, while the blue dots represent the ASSIST synthesized designs. Table 2 zooms in to the top-3 designs found by ASSIST for optimizing cycle time, area and execution time, respectively. We observe that: (1) while the majority of the ASSIST-generated design points have poor QoR, some of the ASSIST-synthesized designs can match or even exceed the QoR of the manual designs when considering a single optimization target such as cycle time or total execution time; (2) certain ASSIST-synthesized designs appear on the Pareto frontier of the three-dimensional QoR tradeoff curve, which shows that ASSIST can effectively explore the design space and discover promising design points with varying QoR characteristics.

The advantage over the manually optimized designs is not because ASSIST can come up with better micro-architecture for a particular pipeline schedule; It is rather because ASSIST facilitates the automatic exploration of processor pipelines with various micro-architectural design choices, and uncovers promising design points under the specific technology target.

4.2 Constrained Performance Optimization

We use the pipeline schedule autotuner to find optimized pipeline schedules that minimizes the total execution time (the runtime for finishing the seven RISC-V benchmarks) under cycle time constraints. Table 3 summarizes the optimization results targeting the Virtex-7 FPGA. To minimize the effect of randomness, we repeat 100 independent experiments for each value of the constraint, where each experiment is allocated an autotuning runtime budget of exploring 16 design points. The runtime of the ASSIST synthesis flow is negligible when compared to the long runtime of the CAD implementation tool. The `Occ.` column in the tables represents the number of occurrences that the specific design point was found during the autotuning process across 100 runs. The pipeline schedule autotuner finds design points with a shorter execution time than the manual designs while satisfying the cycle time constraint, showing the pipeline schedule autotuner can effectively navigate through the design space and identify promising design points satisfying the user-defined constraints and objective.

4.3 Instruction and Data Memories

While ASSIST does not directly synthesize the instruction and data memories, it is designed such that the generated processor pipelines can be easily integrated with various types of memories

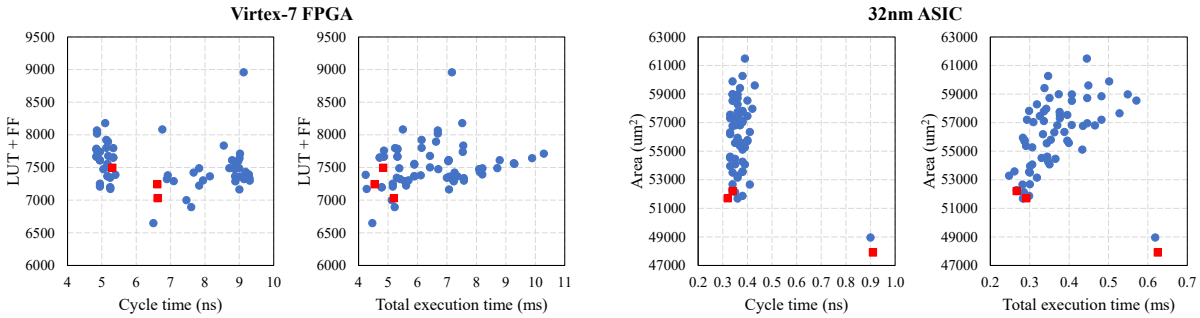

Figure 6: Design space of synthesized pipeline processors targeting the Virtex-7 FPGA — blue dots are designs generated from ASSIST with one to seven pipeline stages. Red dots are the three manually optimized 1-, 2-, and 5-stage designs from [7].

Table 3: Cycle time constraint runtime optimization targeting Virtex-7 FPGA — `T`: cycle time constraint in nanosecond; `Base`: manual 2- and 5-stage designs; `Occ.`: occurrence of the specific designs found during 100 experiment runs; `CT`: clock period in nanosecond; `RT`: total runtime in millisecond over seven kernels; `LUT`: number of lookup tables; `FF`: number of flip-flops.

	Schedule	Occ.	CT	RT	LUT	FF
	Base-2		6.62	5.202	4571	2460
	Base-5		5.31	4.838	4701	2794
T=7.0	[1112222]	41	5.40	4.237	4850	2539
	[1111111]	17	6.50	4.466	4300	2350
	[1112223]	11	5.25	4.271	4465	2707
T=5.0	[1112233]	24	4.86	4.859	4915	2750
	[1112333]	14	4.86	4.859	5013	2747
	[1112334]	11	4.95	5.160	4295	2915

and caches. We study integrating ASSIST pipelines with scratchpad memories as instruction and data memories on the Virtex-7 FPGA. The scratchpad memories are designed as parameterized RTL templates, which are incorporated to the processor pipelines during the optimization process and implemented in the downstream CAD flow. The instruction and data scratchpads are set to 64 words and 128 words for this specific experiment, respectively. Table 4 shows the QoR of the top-3 designs with instruction and data memories found using the pipeline schedule autotuner targeting cycle time minimization with an autotuning runtime budget of exploring 16 design points. We observe that ASSIST generated designs achieve better cycle time than the manually-optimized designs at the cost of small increase in resource usage. This is primarily because the autotuning process optimizes the processor pipeline and the memory subsystem in a holistic way, while the manual designs do not have the physical-level timing information at the design stage.

Table 4: Optimization results with scratchpad memories targeting Virtex-7 FPGA — `Base`: manual 2- and 5-stage designs; `CT`: clock period in nanosecond; `RT`: total runtime in millisecond over seven kernels; `LUT`: number of lookup tables; `FF`: number of flip-flops; `BRAM`: number of block RAMs.

Schedule	CT	RT	LUT	FF	BRAM
Base-2	6.90	5.422	4412	2462	1
Base-5	6.41	5.840	4798	2800	1
[1223333]	6.39	5.123	5106	2685	1
[1123333]	6.39	5.127	5283	2700	1
[1234444]	6.42	4.664	5114	2542	1

4.4 Case Studies

As detailed in Section 3, ASSIST provides a convenient interface for designers to incorporate custom instructions and/or custom instruction set to the baseline RISC-V processor. We present two case studies on instruction extensions of a single custom instruction, followed by an additional case study of implementing a cryptographic instruction set extending the RISC-V 32I base ISA. We target the Virtex-7 FPGAs in these case studies; designs targeting ASICs follow a similar process.

We start by presenting two case studies that add a single custom instruction in each case in the domains of cryptography and machine learning, respectively. The first case study implements the instruction `mixColumns`, which computes one iteration of the loop in the `aes_mixColumns` function (shown in Figure 7) in the Advanced Encryption Standard (AES) kernel. The `mixColumns` instruction takes four 8-bit data, denoted as `buf`, and updates them in-place using XOR, AND, and shift operations. By specializing the loop body of the `aes_mixColumns` function (50 assembly instructions) into a single custom instruction, the dynamic instruction count of the program significantly reduces, leading to an improved QoR. We implement the instruction `binConv` in the second case study. `binConv` executes a 3-by-3 binarized convolution, which is a frequently-used kernel in binarized neural networks. Figure 8 details its original implementation. In the customized processor, the entire procedure call to `conv3x3b` is replaced by a single `binConv` instruction, which significantly reduces the dynamic instruction count. Table 5 summarizes the QoR of the two custom instructions in two separate experiments. The execution time in these two experiments are measured by running the AES kernel and the binarized convolution kernel, respectively, where the custom instructions are incorporated into the benchmarks as inline assembly instructions. When compared to the baseline processor, the ASSIST-synthesized designs require significantly shorter execution time due to the large reduction of dynamic instruction count in critical loops, at the cost of modest increase in cycle time and resource usage.

```
1  uint8_t rj_xtime(uint8_t x) {
2    return (x & 0x80) ? ((x << 1) ^ 0x1b) : (x << 1);}
3
4  void aes_mixColumns(uint8_t *buf) {
5    register uint8_t i, a, b, c, d, e;
6    mix : for (i = 0; i < 16; i += 4) {
7      a = buf[i]; b = buf[i + 1]; c = buf[i + 2];  d = buf[i + 3];
8      e = a ^ b ^ c ^ d;
9      buf[i] ^= e ^ rj_xtime(a^b); buf[i+1] ^= e ^ rj_xtime(b^c);
10     buf[i+2] ^= e ^ rj_xtime(c^d); buf[i+3] ^= e ^ rj_xtime(d^a);
11 }
```

Figure 7: Code snippet of the `mixColumns` function.

```
1  unsigned int conv3x3b(
2    unsigned int buffer[BUFFER_ROWS][BUFFER_COLS],
3    const unsigned int conv_params_m,
4    const unsigned int rr, const unsigned int cc) {
5    unsigned int sum = 0; int kr, kc;
6    for (kr = 0; kr < K; ++kr) {
7      for (kc = 0; kc < K; ++kc) {
8        const char bit0 = ((buffer[rr + kr][(cc + kc) / 16] &
9          (1 << (2 * ((cc + kc) % 16)))) > 0);
10       const char bit1 = ((buffer[rr + kr][(cc + kc) / 16] &
11         (1 << ((2 * ((cc + kc) % 16)) + 1))) > 0);
12       const char wt = (conv_params_m &
13         (1 << ((2 - kr) * 3 + (2 - kc))) > 0);
14       char res = ((wt && bit0) != bit1);
15       sum += (unsigned int)res;
16   }}
17   return sum;
18 }
```

Figure 8: Code snippet of the `binConv` function.

ASSIST can also be used to synthesize domain-specific instruction sets with multiple custom instructions. In this study, we use

Table 5: QoR comparisons between the base processor and the customized processors with instruction extensions targeting Vertex-7 FPGA — `Cycle time` = cycle time in `ns`; `LUT` = number of lookup tables; `FF` = number of flip flops; `# Cycle` = number of cycles to complete the given tasks; `Execution time` = execution time in microsecond; `Base` = RISC-V 32I processor; `AES-custom` = RISC-V 32I processor + `mixColumns` instruction; `binConv-custom` = RISC-V 32I processor + `binConv` instruction.

	Base	AES-custom	Base	binConv-custom
Cycle time (ns)	4.78	4.99 (+4.2%)	4.78	5.23 (+9.4%)
LUT	4760	5308 (+11.5%)	4760	4980 (+4.6%)
FF	2912	2913 (+0.03%)	2912	2762 (-5.2%)
# Cycle	11248	5759 (-48.8%)	84987	13488 (-84.1%)
Execution time (μs)	54	29 (-46.6%)	406	71 (-82.6%)

Table 6: QoR comparisons between the base processor and the processor with cryptographic extensions targeting Vertex-7 FPGA— `Cycle time` = cycle time in `ns`; `Cycle count` = estimated number of cycles to execute a 128-bit AES encryption task; `Execution time` = runtime for a 128-bit AES encryption task in microsecond; `LUT` = number of lookup tables; `FF` = number of flip flops; `RISC-V base` = RISC-V 32I processor; `RISC-V + crypto` = RISC-V 32I processor + cryptographic instruction set extension.

	RISC-V base	RISC-V + crypto
Cycle time (ns)	4.78	5.18 (+8.4%)
Dynamic instruction count	9929	984 (-90.1%)
Execution time (μs)	47.4	5.1 (9.3X faster)
LUT	4760	5460 (+14.7%)
FF	2912	2686 (-7.8%)

the Cryptonite processor ISA [5] as our reference, which can express standard cryptographic algorithms such as AES, DES, MD5 and SHA in a more efficient way than general-purpose ISAs. We implement a 32-bit version of the cryptographic ISA extension with seven custom cryptographic instructions, which execute various bit-level operations specialized for cryptographic algorithms. Implementing the seven cryptographic instructions only required around 100 lines of code in ASSIST. Table 6 summarizes the QoR of the ASSIST-synthesized design when compared to the baseline processor with the RISC-V 32I ISA, where we use the pipeline schedule autotuner to optimize for cycle time. We estimate the runtime of the cryptographic processor using the dynamic instruction count, and observe a moderate 8.4% increase in cycle time with negligible resource overhead. The customized processor executes a 128-bit AES encryption task 9.3X faster than the baseline processor, showing the benefit of processors with domain-specific instruction extensions synthesized from ASSIST.

5 RELATED WORK

Besides the configurable core approach discussed in Section 1, another line of research enables processor synthesis from ADLs. Examples includes LISA [19], EXPRESSION [15], and T-spec/T-piper [17]. Processor synthesis using Bluespec [2] follows a similar approach where designers describe the processor using guarded atomic action constructs. A common requirement of these approach is that designers need to manually specify the datapath and the pipeline schedule. Automatic optimization techniques are usually not provided in ADL-based approaches. Mokhov, et al. propose techniques to synthesize microprocessor designs from high-level ISA specifications [16]. However, they mainly focus on supporting ISA extensions, without providing techniques to automatically pipeline the synthesized datapath. Ralf Dreesen proposes an automatic flow to generate processor pipelines from ISA descriptions [10]. However, the proposed technique lacks support for sharing common resources, and uses precomputed component delay models for pipeline scheduling. The generated designs are both significantly larger in area and slower in frequency than the manually optimized counterparts.

6 CONCLUSIONS

We present ASSIST, a behavior-level microprocessor synthesis framework for RISC-V processors from an instruction set specification. We demonstrate the QoRs from ASSIST by exploring the design space of RISC-V 32I processors, and presents case studies of various instruction set extensions and customized ISAs. Further directions include the extension to various micro-architectural features such as sophisticated branch prediction mechanisms and the support for superscalar execution. Additional optimization targets such as energy efficiency can also be explored by integrating power analysis into the autotuning flow. In addition, machine learning based techniques can potentially be used for fast QoR estimation to alleviate the runtime overhead of the autotuning process.

7 ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their insightful comments. This work is supported in part by a Google Faculty Research Award.

REFERENCES

[1] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe. OpenTuner: An Extensible Framework for Program Autotuning. *Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Aug 2014.

[2] Arvind, R. S. Nikhil, D. L. Rosenband, and N. Dave. High-Level Synthesis: An Essential Ingredient for Designing Complex ASICs. *Int'l Conf. on Computer-Aided Design (ICCAD)*, Nov 2004.

[3] K. Asanović and D. A. Patterson. Instruction Sets Should be Free: the Case for RISC-V. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146*, Aug 2014.

[4] S. Borkar and A. A. Chien. The Future of Microprocessors. *Commun. ACM*, 54(5):67–77, May 2011.

[5] R. Buchty, N. Heintze, and D. Oliva. Cryptonite–A Programmable Crypto Processor Architecture for High-Bandwidth Applications. *International Conference on Architecture of Computing Systems*, Mar 2004.

[6] N. K. Choudhary, S. V. Wadhavkar, T. A. Shah, H. Mayukh, J. Gandhi, B. H. Dwiel, S. Navada, H. H. Najaf-abadi, and E. Rotenberg. FabScalar: Composing Synthesizable RTL Designs of Arbitrary Cores within a Canonical Superscalar Template. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2011.

[7] Christopher Celio. The Sodor Processor Collection. https://github.com/ucb-bar/riscv-sodor.

[8] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, K. Gururaj, and G. Reinman. Accelerator-Rich Architectures: Opportunities and Progresses. *Design Automation Conf. (DAC)*, May 2014.

[9] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 30(4):473–491, Apr 2011.

[10] R. Dreesen. Generating Interlocked Instruction Pipelines from Specifications of Instruction Sets. *Intl'l Conf. on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Oct 2012.

[11] R. E. Gonzalez. Xtensa: A Configurable and Extensible Processor. *IEEE Micro*, 20(2):60–70, Mar 2000.

[12] Y. Hara, H. Tomiyama, S. Honda, H. Takada, and K. Ishii. CHStone: A Benchmark Program Suite for Practical C-Based High-Level Synthesis. *Int'l Symp. on Circuits and Systems (ISCAS)*, May 2008.

[13] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, et al. In-Datacenter Performance Analysis of a Tensor Processing Unit. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2017.

[14] C. E. Leiserson and J. B. Saxe. Retiming Synchronous Circuitry. *Algorithmica*, 6(1-6):5–35, Jun 1991.

[15] P. Mishra, A. Shrivastava, and N. Dutt. Architecture Description Language (ADL)-Driven Software Toolkit Generation for Architectural Exploration of Programmable SOCs. *ACM Trans. on Design Automation of Electronic Systems (TODAES)*, 11(3):626–658, Jun 2004.

[16] A. Mokhov, A. Iliasov, D. Sokolov, M. Rykunov, A. Yakovlev, and A. Romanovsky. Synthesis of Processor Instruction Sets from High-Level ISA Specifications. *IEEE Transactions on Computers*, 63(6):1552–1566, Jun 2014.

[17] E. Nurvitadhi, J. C. Hoe, T. Kam, and S.-L. L. Lu. Automatic Pipelining from Transactional Datapath Specifications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(3):441–454, Mar 2011.

[18] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, et al. A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services. *Int'l Symp. on Computer Architecture (ISCA)*, Oct 2014.

[19] O. Schliebusch, A. Hoffmann, A. Nohl, G. Braun, and H. Meyr. Architecture Implementation using the Machine Description Language LISA. *Asia and South Pacific Design Automation Conf. (ASP-DAC)*, Jan 2002.

[20] Xilinx Inc. *Vivado Design Suite User Guide: High-Level Synthesis*, 2017.

[21] Q. Zhu, O. Shacham, A. Meixner, J. R. Redgrave, D. F. Finchelstein, D. Patterson, N. Desai, D. Stark, E. T. Chang, W. R. Mark, et al. Architecture for High Performance, Power Efficient, Programmable Image Processing, May 2018. US Patent 9,965,824.